CIS 500

Software Foundations

Fall 2005

Programming with OCaml

Functional programming with OCaml

OCaml and this course

The material in this course is mostly conceptual and mathematical. However, experimenting with small implementations is an excellent way to deepen intuitions about many of the concepts we will encounter. For this purpose, we will use the OCaml language.

OCaml is a large and powerful language. For our present purposes, though, we can concentrate just on the “core” of the language, ignoring most of its features.

Functional Programming

OCaml is a functional programming language — i.e., a language in which the functional programming style is the dominant idiom. Other well-known functional languages include Lisp, Scheme, Haskell, and Standard ML.

The functional style can be described as a combination of...

- persistent data structures (which, once built, are never changed)
- recursion as a primary control structure
- heavy use of higher-order functions (functions that take functions as arguments and/or return functions as results)

Imperative languages, by contrast, emphasize

- mutable data structures
- looping rather than recursion
- first-order rather than higher-order programming (though many object-oriented "design patterns" involve higher-order idioms—e.g., Subscribe/Notify, Visitor, etc.)

Computing with Expressions

OCaml is an expression language. A program is an expression. The "meaning" of the program is the value of the expression.

```
# 16 + 18;;
- : int = 34

# 2*8 + 3*6;;
- : int = 34
```

The top level

OCaml provides both an interactive top level and a compiler that produces standard executable binaries. The top level provides a convenient way of experimenting with small programs.

The mode of interacting with the top level is typing in a series of expressions; OCaml evaluates them as they are typed and displays the results (and their types). In the interaction above, lines beginning with `#` are inputs and lines beginning with `-` are the system's responses. Note that inputs are always terminated by a double semicolon.

Giving things names

The `let` construct gives a name to the result of an expression so that it can be used later.

```
# let inchesPerMile = 12*3*1760;;
val inchesPerMile : int = 63360

# let x = 1000000 / inchesPerMile;;
val x : int = 15
```

Functions

```
# let cube (x:int) = x*x*x;;
val cube : int -> int = <fun>

# cube 9;;
- : int = 729
```

We call `x` the parameter of the function `cube`; the expression `x*x*x` is its body.

The expression `cube 9` is an application of `cube` to the argument `9`.

The type printed by OCaml, `int->int` (pronounced "`int` arrow `int`") indicates that `cube` is a function that should be applied to a single, integer argument and that returns an integer.

The type annotation on the parameter `(x:int)` is optional. OCaml can figure it out. However, your life will be much simpler if you put it on.

Note that OCaml responds to a function declaration by printing just `<fun>` as the function's "value."

Here is a function with two parameters:

```
# let sumsq (x:int) (y:int)  = x*x + y*y;;
val sumsq : int -> int -> int = <fun>

# sumsq 3 4;;
- : int = 25
```

The type printed for `sumsq` is `int->int->int`, indicating that it should be applied to two integer arguments and yields an integer as its result.

Note that the syntax for invoking function declarations in OCaml is slightly different from languages in the C/C++/Java family: we write `cube 3` and `sumsq 3 4` rather than `cube(3)` and `sumsq(3,4)`.

The type boolean

There are only two values of type `boolean`: `true` and `false`.

Comparison operations return boolean values.

```
# 1 = 2;;
- : bool = false

# 4 >= 3;;
- : bool = true
```

`not` is a unary operation on booleans.

```
# not (5 <= 10);;
- : bool = false

# not (2 = 2);;
- : bool = false
```

Conditional expressions

The result of the conditional expression `if B then E1 else E2` is either the result of `E1` or that of `E2`, depending on whether the result of `B` is `true` or `false`.

```
# if 3 < 4 then 7 else 100;;
- : int = 7

# if 3 < 4 then (3 + 3) else (10 * 10);;
- : int = 6

# if false then (3 + 3) else (10 * 10);;
- : int = 100

# if false then false else true;;
- : bool = true
```

Lists

One handy structure for storing a collection of data values is a list. Lists are provided as a built-in type in OCaml and a number of other popular languages.

We can build a list in OCaml by writing out its elements, enclosed in square brackets and separated by semicolons.

```
# [1; 3; 2; 5];;
- : int list = [1; 3; 2; 5]
```

The type that OCaml prints for this list is pronounced either "integer list" or "list of integers".

The empty list, written `[]`, is sometimes called "nil."

The types of lists

We can build lists whose elements are drawn from any of the basic types (`int`, `bool`, etc.).

```
# ["cat"; "dog"; "gnu"];;
- : string list = ["cat"; "dog"; "gnu"]

# [true; true; false];;
- : bool list = [true; true; false]
```

We can also build lists of lists:

```
# [[1; 2]; [2; 3; 4]; [5]];;
- : int list list = [[1; 2]; [2; 3; 4]; [5]]
```

In fact, for every type `t`, we can build lists of type `t list`.

Lists are homogeneous

OCaml does not allow different types of elements to be mixed within the same list:

```
# [1; 2; "dog"];;
Characters 7-13:
This expression has type string list but is here used
with type int list
```

Constructing Lists

OCaml provides a number of built-in operations that return lists. The most basic one creates a new list by adding an element to the front of an existing list. It is written `::` and pronounced "cons" (because it constructs lists).

```
# 1 :: [2; 3];;
- : int list = [1; 2; 3]

# let add123 (l: int list) = 1 :: 2 :: 3 :: l;;
val add123 : int list -> int list = <fun>

# add123 [5; 6; 7];;
- : int list = [1; 2; 3; 5; 6; 7]

# add123 [];;
- : int list = [1; 2; 3]
```

Some recursive functions that generate lists

```
# let rec repeat (k:int) (n:int) =  (* A list of n copies of k *)
    if n = 0 then []
    else k :: repeat k (n-1);;

# repeat 7 12;;
- : int list = [7; 7; 7; 7; 7; 7; 7; 7; 7; 7; 7; 7]

# let rec fromTo (m:int) (n:int) = (* The numbers from m to n *)
    if n < m then []
    else m :: fromTo (m+1) n;;

# fromTo 9 18;;
- : int list = [9; 10; 11; 12; 13; 14; 15; 16; 17; 18]
```

Constructing Lists

Any list can be built by "consing" its elements together:

```
-# 1 :: 2 :: 3  :: 2 :: 1 :: [] ;;;
- : int list = [1; 2; 3; 2; 1]
```

In fact,

$$[\ x_1;\ x_2;\ \ldots;\ x_n\ ]$$

is simply a shorthand for

$$x_1\ ::\ x_2\ ::\ \ldots\ ::\ x_n\ ::\ []$$

Note that, when we omit parentheses from an expression involving several uses of `::`, we associate to the right—i.e., `1::2::3::[]` means the same thing as `1::(2::(3::[]))`. By contrast, arithmetic operators like `+` and `-` associate to the left: `1-2-3-4` means `((1-2)-3)-4`.

Taking Lists Apart

OCaml provides two basic operations for extracting the parts of a list.

- `List.hd` (pronounced "head") returns the first element of a list.

```
# List.hd [1; 2; 3];;
- : int = 1
```

- `List.tl` (pronounced "tail") returns everything but the first element.

```
# List.tl [1; 2; 3];;
- : int list = [2; 3]
```

```
# List.tl (List.tl [1; 2; 3]);;
- : int list = [3]

# List.tl (List.tl (List.tl [1; 2; 3]));;
- : int list = []

# List.hd (List.tl (List.tl [1; 2; 3]));;
- : int = 3

# List.hd [[5; 4]; [3; 2]];;
- : int list = [5; 4]

# List.hd (List.hd [[5; 4]; [3; 2]]);;
- : int = 5

# List.tl (List.hd [[5; 4]; [3; 2]]);;
- : int list = [4]
```

Modules – a brief digression

Like most programming languages, OCaml includes a mechanism for grouping collections of definitions into modules.

For example, the built-in module `List` provides the `List.hd` and `List.tl` functions (and many others). That is, the name `List.hd` really means "the function `hd` from the module `List`."

Recursion on lists

Lots of useful functions on lists can be written using recursion. Here's one that sums the elements of a list of numbers:

```
# let rec listSum (l:int list) =
    if l = [] then 0
    else List.hd l + listSum (List.tl l);;

# listSum [5; 4; 3; 2; 1];;
- : int = 15
```

Consing on the right

```
# let rec snoc (l: int list) (x: int) =
    if l = [] then x::[]
    else List.hd l  :: snoc(List.tl l) x;;
val snoc : int list -> int -> int list = <fun>

# snoc [5; 4; 3; 2] 1;;
- : int list = [5; 4; 3; 2; 1]
```

Basic Pattern Matching

Lists can either be empty or non-empty. OCaml provides a convenient pattern-matching construct that determines whether this list is empty, and if it is not, allow access to the first element.

```
# let rec listSum (l: int list) =
    match l with
      []    -> 0
    | x::y -> x + listSum y;;

# listSum [5; 4; 3; 2; 1];;
- : int = 15
```

Pattern matching can be used with types other than lists. For example, here it is used on integers:

```
# let rec fact (n:int) =
    match n with
      0 -> 1
    | _ -> n * fact(n-1);;
```

The `_` pattern here is a wildcard that matches any value.

Complex Patterns

The basic elements (constants, variable binders, wildcards, `[]`, `::`, etc.) may be combined in arbitrarily complex ways in `match` expressions:

```
# let silly (l:int list) =
    match l with
      [_;_;_] -> "three elements long"
    | _::x::y::_::_::rest -> if x>y then "foo" else "bar"
    | _ -> "dunno";;
val silly : int list -> string = <fun>

# silly [1;2;3];;
- : string = "three elements long"
# silly [1;2;3;4];;
- : string = "dunno"
# silly [1;2;3;4;5];;
- : string = "bar"
```

Example: Finding words

Suppose we want to take a list of characters and return a list of lists of characters, where each element of the final list is a "word" from the original list.

```
# split ['t';'h';'e';' ';'b';'r';'o';'w';'n';' ';'d';'o';'g'];;
- : char list list =
      [['t'; 'h'; 'e']; ['b'; 'r'; 'o'; 'w'; 'n']; ['d'; 'o'; 'g']]
```

(Note that character constants are written with single quotes.)

An implementation of split

```
# let rec loop (w:char list) (l:char list) =
    match l with
      [] -> [w]
    | (' '::ls) -> w :: (loop [] ls)
    | (c::ls) -> loop (w @ [c]) ls;;
val loop : char list -> char list -> char list list = <fun>

# let split (l:char list) = loop [] l;;
val split : char list -> char list list = <fun>
```

Aside: Local function definitions

The `loop` function is completely local to `split`: there is no reason for anybody else to use it — or even, for anybody else to be able to see it! It is good style in OCaml to write such definitions as local bindings:

```
# let split (l:char list) =
    let rec loop (w:char list) (l:char list) =
      match l with
        [] -> [w]
      | (' '::ls) -> w :: (loop [] ls)
      | (c::ls) -> loop (w @ [c]) ls
    in
    loop [] l;;
```

In general, any let definition that can appear at the top level

```
# let ...;;
# e;;;
```

can also appear in a `let...in...` form.

```
# let ... in e;;;
```

Tuples

Items connected by commas are “tuples”

```
# "age", 44;;
- : string * int = "age", 44
# ("professor",("age", 33));;
- : string * (string * int) = "professor", ("age", 33)
# ("children", ["bob";"ted";"alice"]);;
- : string * string list = "children", ["bob"; "ted"; "alice"]
```

```
# let g ((x:int),(y:int)) = x*y;;
val g : int * int -> int = <fun>
```

How many arguments does `g` take?

Tuples are not lists

Please do not confuse them!

```
# let tuple = "cow", "dog", "sheep";;
val tuple : string * string * string = "cow", "dog", "sheep"
# let list = ["cow"; "dog"; "sheep"];;
val list : string list = ["cow"; "dog"; "sheep"]
# List.hd tuple;;
This expression has type string * string * string but is here used
with type 'a list
# List.hd list;;
- : string = "cow"
# let tup2 = 1, "cow";;
val tup2 : int * string = 1, "cow"
# let l2 = [1; "cow"];;
This expression has type string but is here used with type int
```

Tuples and Pattern Matching

Tuples can be “deconstructed” by pattern matching:

```
# let lastName name =
    match name with
      (n1,_,_) -> n1;;
# lastName  ("Weirich", "Stephanie", "Penn");;
- : string = "Weirich"
```

Basic Exceptions

OCaml's exception mechanism is roughly similar to that found in, for example, Java.

We begin by defining an exception:

```
# exception Bad;;
```

Now, encountering `raise Bad` will immediately terminate evaluation and return control to the top level:

```
# let rec fact (n:int) =
    if n<0 then raise Bad
    else if n=0 then 1
    else n * fact(n-1);;
# fact (-3);;
Exception: Bad.
```

Naturally, exceptions can also be caught within a program (using the `try...with...` form), but let's leave that for another day.

Data Types

We have seen a number of data types:

`int`

`bool`

`string`

`char`

`lists`

`tuples`

OCaml has a few other built-in data types — in particular, `float`, with operations like `+.`, `*.`, etc.

One can also create completely new data types.

The need for new types

The ability to construct new types is an essential part of most programming languages.

Suppose we are building a (very simple) graphics program that displays circles and squares. We can represent each of these with three real numbers.

A circle is represented by the co-ordinates of its center and its radius. A square is represented by the co-ordinates of its bottom left corner and its width. So we can represent both shapes as elements of the type:

```
float * float * float
```

However, there are two problems with using this type to represent circles and squares. First, it is a bit long and unwieldy, both to write and to read. Second, because their types are identical, there is nothing to prevent us from mixing circles and squares. For example, if we write

```
# let areaOfSquare ((_,_,d):float*float*float) = d *. d;;
```

we might accidentally apply the `areaOfSquare` function to a circle and get a nonsensical result.

(Numerical operations on the `float` type are written differently from the corresponding operations on `int` — e.g., `+.` instead of `+`. See the OCaml manual for more information.)

Data Types

We can improve matters by defining `square` as a new type:

```
# type square = Square of float * float * float;;
```

This does two things:

- It creates a new type called `square` that is different from any other type in the system.
- It creates a constructor called `Square` (with a capital `S`) that can be used to create a `square` from three floats. For example:

```
# Square(1.1,2.2,3.3);;
- : square = Square (1.1, 2.2, 3.3)
```

Taking data types apart

We take types apart with (surprise, surprise...) pattern matching.

```
# let areaOfSquare (s:square) =
    match s with
      Square(_, _, d) -> d *. d;;
val areaOfSquare : square -> float = <fun>

# let bottomLeftCoords (s:square) =
    match s with
      Square(x, y, _) -> (x,y);;
val bottomLeftCoords : square -> float * float = <fun>
```

So we can use constructors like `Square` both as functions and as patterns.

Constructors are recognized by being capitalized (the first letter is upper case).

These functions can be written a little more concisely by combining the pattern matching with the function header:

```
# let areaOfSquare (Square(_, _, d):square) = d *. d;;
# let bottomLeftCoords (Square(x, y, _):square) = (x,y);;
```

Continuing, we can define a data type for circles in the same way.

```
# type circle = Circle of float * float * float;;

# let c = Circle (1.0, 2.0, 2.0);;

# let areaOfCircle (Circle(_, _, r):circle) = 3.14159 *. r *. r;;

# let centerCoords (Circle(x, y, _):circle) = (x,y);;

# areaOfCircle c;;
- : float = 12.56636
```

We cannot now apply a function intended for type `square` to a value of type `circle`:

```
# areaOfSquare(c);;
This expression has type circle but is here used with type square.
```

Variant types

Going back to the idea of a graphics program, we obviously want to have several shapes on the screen at once. For this we'd probably want to keep a list of circles and squares, but such a list would be heterogenous. How do we make such a list?

The solution is to build a type that can be either a circle or a square.

```
# type shape = Circle of float * float * float
             | Square of float * float * float;;
```

Now both constructors `Circle` and `Square` create values of type `shape`. For example:

```
# Square (1.0, 2.0, 3.0);;
- : shape = Square (1.000000, 2.000000, 3.000000)
```

A type that can have more than one form is often called a variant type.

We can also write functions that do the right thing on all forms of a variant type. Again we use pattern matching:

```
# let area (s:shape) =
    match s with
      Circle (_, _, r) -> 3.14159 *. r *. r
    | Square (_, _, d) -> d *. d;;

# area (Circle (0.0, 0.0, 1.5));;
- : float = 7.0685775
```

A “heterogeneous” list:

```
# let l = [Circle (0.0, 0.0, 1.5); Square (1.0, 2.0, 1.0);
           Circle (2.0, 0.0, 1.5); Circle (5.0, 0.0, 2.5)];;
```

Mixed-mode Arithmetic

Many programming languages (Lisp, Basic, Perl, database query languages) use variant types internally to represent numbers that can be either integers or floats. This amounts to "tagging" each numeric value with an indicator that says what kind of number it is.

```
# type num = Int of int | Float of float;;

# let add (r1:num) (r2:num) =
    match (r1,r2) with
      (Int i1,   Int i2)   -> Int (i1 + i2)
    | (Float r1, Int i2)   -> Float (r1 +. float i2)
    | (Int i1,   Float r2) -> Float (float i1 +. r2)
    | (Float r1, Float r2) -> Float (r1 +. r2);;

# add (Int 3) (Float 4.5);;
- : num = Float 7.5
```

Multiplication, `mult` follows exactly the same pattern:

```
# let mult (r1:num) (r2:num) =
    match (r1,r2) with
      (Int i1,   Int i2)   -> Int (i1 * i2)
    | (Float r1, Int i2)   -> Float (r1 *. float i2)
    | (Int i1,   Float r2) -> Float (float i1 *. r2)
    | (Float r1, Float r2) -> Float (r1 *. r2);;
```

A Data Type for Optional Values

Suppose we are implementing a simple lookup function for a telephone directory. We want to give it a string and get back a number (say an integer). We expect to have a function `lookup` whose type is

```
lookup: string  -> directory -> int
```

where `directory` is a (yet to be decided) type that we'll use to represent the directory.

However, this isn't quite enough. What happens if a given string isn't in the directory? What should `lookup` return?

There are several ways to deal with this issue. One is to raise an exception. Another is based on the following data type:

```
# type maybe = Absent | Present of int;;
```

To see how this type is used, let's represent our directory as a list of pairs:

```
# type directory = (string * int) list ;;
# let directory  = [("Joe", 1234); ("Martha", 5672);
                    ("Jane", 3456); ("Ed", 7623)];;

# let rec lookup (s:string) (l:directory) =
    match l with
      [] -> Absent
    | (k,i)::t -> if k = s then Present(i)
                            else lookup s t;;

# lookup "Jane" directory;;
- : maybe = Present 3456

# lookup "Karen" directory;;
- : maybe = Absent
```

Built-in options

Because options are often useful in functional programming, OCaml provides a built-in type `t option` for each type `t`. Its constructors are `None` (corresponding to `Absent`) and `Some` (for `Present`).

```
# let rec lookup (s:string) (l:directory) =
    match l with
      [] -> None
    | (k,i)::t -> if k = s then Some(i)
                           else lookup s t;;

# lookup "Jane" directory;;
- : int option = Some 3456
```

Enumerations

Our `maybe` data type has one variant, `Absent`, that is a "constant" constructor carrying no data values with it. Data types in which all the variants are constants can actually be quite useful...

```
# type color = Red | Yellow | Green;;
# let next (c:color) =
    match c with Green -> Yellow | Yellow -> Red | Red -> Green;;
```

```
# type day =   Sunday | Monday | Tuesday | Wednesday
             | Thursday | Friday | Saturday;;
# let weekend (d:day) =
    match d with
      Saturday -> true
    | Sunday   -> true
    | _        -> false;;
```

A Boolean Data Type

A simple data type can be used to replace the built-in booleans.

We use the constant constructors `True` and `False` to represent true and false. We'll use different names as needed to avoid confusion between our booleans and the built-in ones:

```
# type myBool = False | True;;
# let myNot (b:myBool) = match b with False -> True | True -> False;;
# let myAnd (b1:myBool) (b2:myBool) =
    match (b1,b2) with
      (True,  True)  ->  True
    | (True,  False) ->  False
    | (False, True)  ->  False
    | (False, False) ->  False;;
```

Note that the behavior of `myAnd` is not quite the same as the built-in `&&`!

Recursive Types

Consider the tiny language of arithmetic expressions defined by the following grammar:

```
exp  ::=  number
          exp + exp
          exp - exp
          exp * exp
```

We can translate this grammar directly into a datatype definition:

```
type ast =
    ANum of int
  | APlus of ast * ast
  | AMinus of ast * ast
  | ATimes of ast * ast;;
```

Notes:

- This datatype (like the original grammar) is recursive.
- The type `ast` represents abstract syntax trees, which capture the underlying tree structure of expressions, suppressing surface details such as parentheses

An evaluator for expressions

Goal: write an evaluator for these expressions.

```
val eval : ast -> int = <fun>

# eval (ATimes (APlus (ANum 12, ANum 340), ANum 5));;
- : int = 1760
```

The solution uses a recursive function plus a pattern match.

```
let rec eval (e:ast) =
  match e with
    ANum i -> i
  | APlus (e1,e2) -> eval e1 + eval e2
  | AMinus (e1,e2) -> eval e1 - eval e2
  | ATimes (e1,e2) -> eval e1 * eval e2;;
```

The solution uses a recursive function plus a pattern match.

```
let rec eval (e:ast) =
  match e with
    ANum i -> i
  | APlus (e1,e2) -> eval e1 + eval e2
  | AMinus (e1,e2) -> eval e1 - eval e2
  | ATimes (e1,e2) -> eval e1 * eval e2;;
```

The pattern of recursion follows the definition of the datatype.

```
type ast =
    ANum of int
  | APlus of ast * ast
  | AMinus of ast * ast
  | ATimes of ast * ast;;
```

Polymorphism

```
# let rec last l =
    match l with
      []    -> raise Bad
    | [x]   -> x
    | _::y -> last y
```

Polymorphism

```
# let rec last l =
    match l with
      []   -> raise Bad
    | [x]  -> x
    | _::y -> last y
```

What type should we give to the parameter `l`?

Polymorphism

```
# let rec last l =
    match l with
      []   -> raise Bad
    | [x]  -> x
    | _::y -> last y
```

What type should we give to the parameter `l`?

It doesn't matter what type of objects are stored in the list, we could make it `int list` or `bool list` and OCaml would not complain. However, if we chose one of these types, would not be able to apply `last` to the other.

OCaml lets us use a type variable to abstract part of a type if it is not important. We can give `l` the type `'a list` (pronounced "alpha"), standing for an arbitrary type. When we use the function, OCaml will figure out what type we need.

Polymorphism

This version of `last` is said to be polymorphic, because it can be applied to many different types of arguments. ("Poly" = many, "morph" = shape.)

Note that the type of the elements of `l` is `'a` (pronounced "alpha"). This is a type variable, which can instantiated, each time we apply `last`, by replacing `'a` with any type that we like. The instances of the type `'a list -> 'a` include

```
int list -> int
string list -> string
int list list -> int list
etc.
```

In other words,

```
last : 'a list -> 'a
```

can be read, "`last` is a function that takes a list of elements of any type alpha and returns an element of alpha."

A polymorphic append

```
# let rec append (l1: 'a list) (l2: 'a list) =
    match l1 with
      [] -> l2
    | (hd::tl) -> hd :: append tl l2;;
val append : 'a list -> 'a list -> 'a list = <fun>

# append [4; 3; 2]  [6; 6; 7];;
- : int list = [4; 3; 2; 6; 6; 7]

# append ["cat"; "in"] ["the"; "hat"];;
- : string list = ["cat"; "in"; "the"; "hat"]
```

A polymorphic rev

```
# let rec revaux (l: 'a list) (res: 'a list) =
    match l with
      [] -> res
    | (hd::tl) -> revaux tl (hd :: res) ;;
val revaux : 'a list -> 'a list -> 'a list = <fun>

# let rev (l: 'a list) = revaux l [];;
val rev : 'a list -> 'a list = <fun>

# rev ["cat"; "in"; "the"; "hat"];;
- : string list = ["hat"; "the"; "in"; "cat"]

# rev [false; true];;
- : bool list = [true; false]
```

Polymorphic `repeat`

```
# let rec repeat (k:'a) (n:int) =  (* A list of n copies of k *)
    if n = 0 then []
    else k :: repeat k (n-1);;

# repeat 7 12;;
- : int list = [7; 7; 7; 7; 7; 7; 7; 7; 7; 7; 7; 7]

# repeat true 3;;
- : bool list = [true; true; true]

# repeat [6;7] 4;;
- : int list list = [[6; 7]; [6; 7]; [6; 7]; [6; 7]]
```

What is the type of `repeat`?

map: “apply-to-each”

OCaml has a predefined function `List.map` that takes a function `f` and a list `l` and produces another list by applying `f` to each element of `l`. We’ll soon see how to define `List.map`, but first let’s look at some examples.

```
# List.map square [1; 3; 5; 9; 2; 21];;
- : int list = [1; 9; 25; 81; 4; 441]

#  List.map not [false; false; true];;
- : bool list = [true; true; false]
```

Note that `List.map` is polymorphic: it works for lists of integers, strings, booleans, etc.

More on map

An interesting feature of `List.map` is its first argument is itself a function. For this reason, we call `List.map` a higher-order function.

Natural uses for higher-order functions arise frequently in programming. One of OCaml's strengths is that it makes higher-order functions very easy to work with.

In other languages such as Java, higher-order functions can be (and often are) simulated using objects.

filter

Another useful higher-order function is `List.filter`. When applied to a list `l` and a boolean function `p`, it extracts from `l` the list of those elements for which `p` returns `true`.

```
# let rec even (n:int) =
    if n=0 then true
    else if n=1 then false
    else if n<0 then even (-n)
    else even (n-2);;
val even : int -> bool = <fun>

# List.filter even [1; 2; 3; 4; 5; 6; 7; 8; 9];;
- : int list = [2; 4; 6; 8]

# List.filter palindrome  [[1]; [1; 2; 3]; [1; 2; 1]; []];;
- : int list list = [[1]; [1; 2; 1]; []]
```

Note that, like map, `List.filter` is polymorphic—it works on lists of any type.

Defining `map`

`List.map` comes predefined in the OCaml system, but there is nothing magic about it—we can easily define our own `map` function with the same behavior.

```
let rec map (f: 'a->'b) (l: 'a list) =
      match l with
        [] -> []
      | (hd::tl) -> f hd :: map f tl ;;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

The type of `map` is probably even more polymorphic than you expected! The list that it returns can actually be of a different type from its argument:

```
# map String.length ["The"; "quick"; "brown"; "fox"];;
- : int list = [3; 5; 5; 3]
```

Defining `filter`

Similarly, we can define our own `filter` that behaves the same as `List.filter`.

```
let rec filter (p: 'a->bool) (l: 'a list) =
      match l with
         [] -> []
      | (hd::tl) -> if p hd then hd :: filter p tl
                    else filter p tl ;;
val filter : ('a -> bool) -> 'a list -> 'a list = <fun>
```

Generic Programming

The polymorphism in ML that arises from type parameters is an example of generic programming. (`mapl`, `filter`, etc.) are good examples of generic functions. Different languages support generic programming in different ways...

- parametric polymorphism allows functions to work uniformly over arguments of different types.E.g., `last : 'a list -> 'a`
- ad hoc polymporphism (or overloading) allows an operation to behave in different ways when applied to arguments of different types. There is no such polymorphism in OCaml, but most languages allow some overloading (e.g.`2+3` and `2.4 +3.6`).
- subtype polymporphism allows operations to be defined for collections of types sharing some common structure

  e.g., a `feed` operation might make sense for values of `animal` and all its "refinements"—`cow`, `tiger`, `moose`, etc.

OCaml supports parametric polymorphism in a very general way, and also supports subtyping (Though we shall not get to see this aspect of OCaml, its support for subtyping is what distinguishes it from other dialects of ML.) It does not allow overloading.

Java provides a subtyping as well as moderately powerful overloading, but no parametric polymorphism. (Java 1.5 beta version has parametric polymorphism, called “Generics”.)

Confusingly, the bare term “polymorphism” is used to refer to parametric polymorphism in the ML community and for subtype polymorphism in the Java community!

Approaches to Typing

- A strongly typed language prevents programs from accessing private data, corrupting memory, crashing the machine, etc.
- A weakly typed language does not.
- A statically typed language performs type-consistency checks at when programs are first entered.
- A dynamically typed language delays these checks until programs are executed.

	Weak	Strong
Dynamic		Lisp, Scheme, Perl, Python, Smalltalk
Static	C, C++	ML, ADA, Java*

*Strictly speaking, Java should be called "mostly static"

Practice with Types

What are the types of the following functions?

- `let f (x:int) = x + 1`
- `let f x = x + 1`
- `let f (x:int) = [x]`
- `let f x = [x]`
- `let f x = x`
- `let f x = hd(tl x) :: [1.0]`
- `let f x = hd(tl x) :: []`
- `let f x = 1 :: x`
- `let f x y = x :: y`

- `let f x y = x :: []`
- `let f x = x @ x`
- `let f x = x :: x`
- `let f x y z = if x>3 then y else z`
- `let f x y z  = if x>3 then y else [z]`

And one more:

```
let rec f x  =
    if (tl x) = [] then x
    else f (tl x)
```

Programming with Functions in OCaml

Functions as Data

Functions in OCaml are first class — they have the same rights and privileges as values of any other types. E.g., they can be

- passed as arguments to other functions
- returned as results from other functions
- stored in data structures such as tuples and lists
- etc.

Multi-parameter functions

We have seen two ways of writing functions with multiple parameters:

```
# let foo x y = x + y;;
val foo : int -> int -> int = <fun>

# let bar (x,y) = x + y;;
val bar : int * int -> int = <fun>
```

The first takes its two arguments separately; the second takes a tuple and uses a pattern to extract its first and second components.

The syntax for applying these two forms of function to their arguments differs correspondingly:

```
# foo 2 3;;
- : int = 5

# bar (4,5);;
- : int = 9

# foo (2,3);;
This expression has type int * int
but is here used with type int

# bar 4 5;;
This function is applied to too many arguments
```

Partial Application

One advantage of the first form of multiple-argument function is that such functions may be partially applied.

```
# let foo2 = foo 2;;
val foo2 : int -> int = <fun>

# foo2 3;;
- : int = 5

# foo2 5;;
- : int = 7

# List.map foo2 [3;6;10;100];;
- : int list = [5; 8; 12; 102]
```

Currying

Obviously, these two forms are closely related — given one, we can easily define the other.

```
# let foo' x y = bar (x,y);;
val foo' : int -> int -> int = <fun>

# let bar' (x,y) = foo x y;;
val bar' : int * int -> int = <fun>
```

Currying

Indeed, these transformations can themselves be expressed as (higher-order) functions:

```
# let curry f x y = f (x,y);;
val curry : ('a * 'b -> 'c) -> 'a -> 'b -> 'c = <fun>

# let foo'' = curry bar;;
val foo'' : int -> int -> int = <fun>

# let uncurry f (x,y) = f x y;;
val uncurry : ('a -> 'b -> 'c) -> 'a * 'b -> 'c = <fun>

# let bar'' = uncurry foo;;
val bar'' : int * int -> int = <fun>
```

A Closer Look

The type `int -> int -> int` can equivalently be written `int -> (int -> int)`.

That is, a function of type `int -> int -> int` is actually a function that, when applied to an integer, yields a function that, when applied to an integer, yields an integer.

Similarly, an application like `foo 2 3` is actually shorthand for `(foo 2) 3`.

Formally: `->` is right-associative and application is left-associative.

Anonymous Functions

It is fairly common in OCaml that we need to define a function and use it just once.

```
# let timesthreeplustwo x = x*3 + 2;;
val timesthreeplustwo : int -> int = <fun>

# List.map timesthreeplustwo [4;3;77;12];;
- : int list = [14; 11; 233; 38]
```

To save making up names for such functions, OCaml offers a mechanism for writing them in-line:

```
# List.map (fun x -> x*3 + 2) [4;3;77;12];;
- : int list = [14; 11; 233; 38]
```

Anonymous Functions

Anonymous functions may appear, syntactically, in the same places as values of any other types.

For example, the following let-bindings are completely equivalent:

```
# let double x = x*2;;
val double : int -> int = <fun>

# let double' = (fun x -> x*2);;
val double' : int -> int = <fun>

# double 5;;
- : int = 10

# double' 5;;
- : int = 10
```

Anonymous Functions

We can even write:

```
# (fun x -> x*2) 5;;
- : int = 10
```

Or (slightly more usefully):

```
# (if 5*5 > 20 then (fun x -> x*2) else (fun x -> x+3)) 5;;
- : int = 10
```

The conditional yields a function on the basis of some boolean test, and its result is then applied to 5.

Quick Check

What is the type of `l`?

```
# let l = [ (fun x -> x + 2);
            (fun x -> x * 3);
            (fun x -> if x > 4 then 0 else 1) ];;
```

Applying a list of functions

```
# let l = [ (fun x -> x + 2);
            (fun x -> x * 3);
            (fun x -> if x > 4 then 0 else 1) ];;
val l : (int -> int) list = [<fun>; <fun>; <fun>]

# let applyto x f = f x;;
val applyto : 'a -> ('a -> 'b) -> 'b = <fun>

# List.map (applyto 10) l;;
- : int list = [12; 30; 0]

# List.map (applyto 2) l;;
- : int list = [4; 6; 1]
```

Another useful higher-order function: `fold`

```
# let rec fold f l acc =
  match l with
    [] -> acc
  | a::l -> f a (fold f l acc);;
val fold : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
```

For example:

```
# fold (fun a b -> a + b) [1; 3; 5; 100] 0;;
- : int = 109
```

In general:

`f [a1; ...; an] b`

is

`f a1 (f a2 (... (f an b) ...)).`

Using `fold`

Most of the list-processing functions we have seen can be defined compactly in terms of `fold`:

```
# let listSum l =
    fold (fun a b -> a + b) l 0;;
val listSum : int list -> int = <fun>

# let length l =
    fold (fun a b -> b + 1) l 0;;
val length : 'a list -> int = <fun>
```

Using `fold`

```
# let map f l =
    fold (fun a b -> (f a) :: b) l [];;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>

# let filter p l =
    fold (fun a b -> if p a then (a::b) else b) l [];;
```

Using `fold`

And even:

```
# (* List of numbers from m to n, as before *)
  let rec fromTo m n =
    if n < m then []
    else m :: fromTo (m+1) n;;
val fromTo : int -> int -> int list = <fun>

# let fact n =
    fold (fun a b -> a * b) (fromTo 1 n) 1;;
val fact : int -> int = <fun>
```

Quick Check

What is the type of this function?

```
# let foo l =
    fold (fun a b -> List.append b [a]) l [];;
```

What does it do?

Forms of `fold`

The OCaml `List` module actually provides two folding functions:

```
List.fold_left  : ('a -> 'b -> 'a) -> 'a      -> 'b list -> 'a
List.fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b      -> 'b
```

The one we're calling `fold` (here and in the homework assignment) is `List.fold_right`.

`List.fold_left` performs the same basic operation but takes its arguments in a different order.

The `unit` type

OCaml provides another built-in type called `unit`, with just one inhabitant, written `()`.

```
# let x = ();;
val x : unit = ()

# let f () = 23 + 34;;
val f : unit -> int = <fun>

# f ();;
- : int = 57
```

Why is this useful?

Uses of `unit`

A function from `unit` to `'a` is a delayed computation of type `'a`.

When we define the function...

```
# let f () = <long and complex calculation>;;
val f : unit -> int = <fun>
```

... the `long and complex calculation` is just boxed up in a closure that we can save for later (by binding it to a variable, e.g.).

When we actually need the result, we apply `f` to `()` and the calculation actually happens:

```
# f ();;
- : int = 57
```

Thunks

A function accepting a `unit` argument is often called a thunk.

Thunks are widely used in functional programming.

A typical example...

Suppose we are writing a function where we need to make sure that some "finalization code" gets executed, even if an exception is raised.

```
# let read file =
    let chan = open_in file in
    try
      let nbytes = in_channel_length chan in
      let string = String.create nbytes in
      really_input chan string 0 nbytes;
      close_in chan;
      string
    with exn ->
      (* finalize channel *)
      close_in chan;
      (* re-raise exception *)
      raise exn;;
```

We can avoid duplicating the finalization code by wrapping it in a thunk:

```
# let read file =
    let chan = open_in file in
    let finalize () = close_in chan in
    try
      let nbytes = in_channel_length chan in
      let string = String.create nbytes in
      really_input chan string 0 nbytes;
      finalize();
      string
    with exn ->
      (* finalize channel *)
      finalize();
      (* re-raise exception *)
      raise exn;;
```

In fact, we can go further...

```
# let unwind_protect body finalize =
    try
      let res = body() in
      finalize();
      res
    with exn ->
      finalize();
      raise exn;;

# let read file =
    let chan = open_in file in
    unwind_protect
      (fun () ->
         let nbytes = in_channel_length chan in
         let string = String.create nbytes in
         really_input chan string 0 nbytes;
         string)
      (fun () -> close_in chan);;
```